

Ficha 3

Programação Imperativa

1 Definição de Tipos

1. Considere o seguinte tipo para representar *stacks* de números inteiros.

```
#define MAX 100
typedef struct stack {
    int sp;
    int valores [MAX];
} STACK;
```

Defina as seguintes funções sobre este tipo:

- (a) `void initStack (STACK *s)` que inicializa uma stack (passa a representar uma stack vazia)
- (b) `int isEmptyS (STACK *s)` que testa se uma stack é vazia
- (c) `int push (STACK *s, int x)` que acrescenta `x` ao topo de `s`; a função deve retornar 0 se a operação for feita com sucesso (i.e., se a stack ainda não estiver cheia) e 1 se a operação não for possível (i.e., se a stack estiver cheia).
- (d) `int pop (STACK *s, int *x)` que remove de uma stack o elemento que está no topo. A função deverá colocar no endereço `x` o elemento removido. A função deverá retornar 0 se a operação for possível (i.e. a stack não está vazia) e 1 em caso de erro (stack vazia).
- (e) `int top (STACK *s, int *x)` que coloca no endereço `x` o elemento que está no topo da stack (sem modificar a stack). A função deverá retornar 0 se a operação for possível (i.e. a stack não está vazia) e 1 em caso de erro (stack vazia).

2. Considere o seguinte tipo para representar *queues* de números inteiros.

```
#define MAX 100
typedef struct queue {
    int inicio, tamanho;
    int valores [MAX];
} QUEUE;
```

Defina as seguintes funções sobre este tipo:

- (a) `void initQueue (QUEUE *q)` que inicializa uma queue (passa a representar uma queue vazia)

- (b) `int isEmptyQ (QUEUE *q)` que testa se uma queue é vazia
 - (c) `int enqueue (QUEUE *q, int x)` que acrescenta `x` ao fim de `q`; a função deve retornar 0 se a operação fôr feita com sucesso (i.e., se a queue ainda não estiver cheia) e 1 se a operação não fôr possível (i.e., se a queue estiver cheia).
 - (d) `int dequeue (QUEUE *q, int *x)` que remove de uma queue o elemento que está no início. A função deverá colocar no endereço `x` o elemento removido. A função deverá retornar 0 se a operação for possível (i.e. a queue não está vazia) e 1 em caso de erro (queue vazia).
 - (e) `int front (QUEUE *q, int *x)` que coloca no endereço `x` o elemento que está no início da queue (sem modificar a queue). A função deverá retornar 0 se a operação for possível (i.e. a queue não está vazia) e 1 em caso de erro (queue vazia).
3. Nas representações de stacks e queues sugeridas nas alíneas anteriores o array de valores tem um tamanho fixo (definido pela constante `MAX`). Uma consequência dessa definição é o facto de as funções de inserção (`pop` e `enqueue`) poderem não ser executadas por se ter excedido a capacidade das estruturas.

Uma definição alternativa consiste em não ter um array com tamanho fixo e sempre que seja preciso mais espaço, realocar o array para um de tamanho superior (normalmente duplica-se o tamanho do array).

Considere então as seguintes definições alternativas e adapte as funções definidas atrás para estas novas representações.

Use as funções `malloc` e `free` cujo tipo está definido em `stdlib.h`.

- (a)

```
typedef struct stack {
    int size; // guarda o tamanho do array valores
    int sp;
    int *valores;
} STACK;
```
- (b)

```
typedef struct queue {
    int size; // guarda o tamanho do array valores
    int inicio, tamanho;
    int *valores;
} QUEUE;
```

4. Para gerir a informação sobre os alunos inscritos a uma dada disciplina, é necessário armazenar os seguintes dados:

- Nome do aluno (string com no máximo 60 caracteres)
- Número do aluno
- Avaliação

Quanto à avaliação, há dois métodos alternativos:

- No método A, a avaliação tem uma componente periódica (6 mini-testes quinzenais), e uma componente final (teste e/ou exame).
- No método B, a avaliação consta apenas do teste e/ou exame final.

Considere que para cada prova, os resultados são armazenados como um número inteiro (de 0 a 100).

- (a) Defina os tipos `Avaliacao`, `Aluno` e `Turma`. Assuma que o número de alunos nunca ultrapassa 100, podendo por isso usar um array para armazenar a informação da turma.
- (b) Defina uma função `int acrescentaAluno (Turma t, Aluno a)` que acrescenta a informação de um dado aluno a uma turma. A função deverá retornar 0 se a operação for feita com sucesso.
- (c) Defina uma função `int procura (Turma t, int numero)` que procura esse aluno na turma. A função deve retornar -1 se a informação desse aluno não existir; caso exista deve retornar o índice onde essa informação se encontra.
- (d) Defina uma função que calcula a nota final (um *float* de 0 a 20) de um aluno.
 - Para alunos do método A, a nota final é a média ponderada da parte periódica (40%) com a nota do teste/exame (60%).
 - A nota da parte periódica é obtida como a média das 5 melhores notas dos mini testes
 - A nota do teste/exame é a nota do teste caso o aluno não tenha realizado o exame; é a nota do exame no outro caso.
- (e) Defina uma função que determine quantos alunos obtiveram aproveitamento à disciplina (nota final maior ou igual a 10).

2 Listas Ligadas

Considere a seguinte definição de um tipo para representar listas ligadas de inteiros.

```
typedef struct slist *LInt;
```

```
typedef struct slist {  
    int valor;  
    LInt prox;  
} Nodo;
```

1. Apresente uma sequência de instruções que coloque na variável `a` do tipo `LInt`, uma lista com 3 elementos: 10, 5 e 15 (por esta ordem).
2. Apresente definições recursivas e iterativas das seguintes funções:
 - (a) `int length (LInt)` que calcula o comprimento de uma lista ligada.
 - (b) `LInt clone (LInt)` que cria uma cópia de uma lista.
 - (c) `void freeL (LInt)` que liberte o espaço ocupado por uma lista.
 - (d) `void imprime (LInt)` que imprime no ecrã os elementos de uma lista (um por linha).
 - (e) `LInt reverse (LInt)` que inverte uma lista (sem criar uma nova lista).

- (f) `LInt insert (LInt, int)` que insere ordenadamente um elemento numa lista ordenada.
 - (g) `LInt remove (LInt, int)` que remove um elemento de uma lista ordenada.
3. Defina uma função `LInt merge (LInt a, LInt b)` que junta duas listas ordenadas numa única lista ordenada. Use essa função para definir uma função que ordena uma lista ligada por *mergesort*.
 4. Defina uma função `void split (LInt l, int x, LInt *mx, LInt *Mx)` que, dada uma lista ligada `l` e um inteiro `x`, parte a lista em duas (retornando os endereços dos primeiros elementos da lista em `*mx` e `*Mx`): uma com os elementos de `l` menores do que `x` e a outra com os restantes. Note que esta função não deverá criar cópias dos elementos da lista.
Use essa função para definir uma função de ordenação de listas ligadas por *quicksort*.
 5. Apresente definições (não recursivas) das seguintes funções, tendo o cuidado de libertar a memória ocupada pelos elementos removidos.
 - (a) `LInt remove (LInt, int)` que remove todas as ocorrências de um dado inteiro de uma lista.
 - (b) `LInt removeDups (LInt)` que remove os valores repetidos de uma lista (deixando apenas a primeira ocorrência).
 - (c) `LInt removeMaior (LInt)` que remove (a primeira ocorrência) o maior elemento de uma lista não vazia.
 6. Apresente definições (preferencialmente não recursivas) para:
 - (a) `LInt init (LInt l)` que remove o último elemento de uma lista não vazia (libertando o correspondente espaço).
 - (b) `LInt snoc (LInt l, int x)` que acrescenta um elemento no fim da lista.
 - (c) `LInt concat (LInt a, LInt b)` que acrescenta a lista `b` a `a`, retornando o início da lista resultante).

As duas últimas funções referidas na alínea anterior são muito pouco eficientes porque obrigam a percorrer uma lista apenas para nos posicionarmos no seu último elemento. Uma forma de melhorarmos a eficiência dessas operações consiste em guardar, para cada lista, dois endereços: o da primeira e o da última componentes.

```
typedef struct difl {
    LInt inicio, fim;
} DifList;
```

Redefina agora as duas operações da alínea anterior usando este novo tipo.

7. `DifList snoc (DifList l, int x)`
8. `DifList concat (DifList a, DifList b)`

Suponha que para resolver o problema descrito na secção 1 se optou por usar uma lista ligada em vez de um array.

9. Defina os novos tipos de dados para esta implementação.
10. Apresente definições das funções `acrescentaAluno`, `procura` e `aprovados` para esta nova implementação.
Tenha o cuidado de rever os tipos destas funções nesta nova implementação.
11. Considere que os dados sobre os alunos e as notas dos minitests vão ser lidos a partir do teclado com o seguinte formato:
 - na primeira linha será lido um inteiro `n` que representa o número de alunos inscritos.
 - a segunda linha contém um inteiro `k` que representa o número de minitests efectuados (este número pode ser maior do que 6 uma vez que a turma pode estar dividida em turnos).
 - de seguida aparece a informação (número e nome) de cada aluno inscrito (um por linha, correspondendo por isso a `n` linhas)
 - de seguida aparecem `k` blocos em que cada bloco tem a seguinte sintaxe:
 - a primeira linha contém o número `p` de notas deste turno/miniteste
 - seguem-se `p` linhas em que cada uma contém o número do aluno e a nota.
 - finalmente aparecem dois blocos sobre as notas do teste e do exame. Cada um destes blocos é constituído por:
 - O número `q` de alunos que foram ao teste/exame
 - `q` linhas com as notas de cada teste/exame, contendo o número do aluno e a classificação.

Escreva um programa que lê a informação com este formato (do teclado) e escreve no ecrã a seguinte informação sobre cada aluno (um aluno por linha);

- número e nome
- método de avaliação (A ou B)
- Nota da avaliação periódica (no caso dos alunos do método A)
- Nota do Exame
- Nota Final

Qualquer elemento de avaliação em falta (miniteste ou exame final) deve ser considerado com o valor 0.

Um aluno é do método A desde que tenha tido avaliação a pelo menos um mini-teste.

Considere a seguinte definição para implementar listas duplamente ligadas de inteiros:

```
typedef struct node *DList;

typedef struct node {
    int value;
    DList prev, next;
} Node;
```

Defina funções de processamento destas listas (nas duas primeiras alíneas assumo que se pretende manter as listas ordenadas por ordem crescente)

12. `DList addInt (DList l, int x)` que acrescenta um elemento à lista.
13. `DList exists (DList l, int x)` que determina se um elemento existe na lista; no caso de existir deve retornar o endereço da correspondente célula; caso contrário deve retornar `NULL`. Comece por definir duas funções `DList lookLeft (DList l, int x)` e `DList lookRight (DList l, int x)` que procuram um elemento para a direita ou para a esquerda.
14. `DList remove (DList l)` que remove um nodo da lista (libertando o correspondente espaço em memória).
15. `DList rewind (DList l)` que retorna o endereço do primeiro nodo da lista (`NULL` caso a lista seja vazia).
16. `DList forward (DList l)` que retorna o endereço do último nodo da lista (`NULL` caso a lista seja vazia).

3 Árvores

1. Considere a seguinte definição de um tipo para representar árvores binárias de inteiros.

```
typedef struct nodo *ABin;
```

```
struct nodo {
    int valor;
    ABin esq, dir;
};
```

- (a) Apresente uma definição recursiva de uma função que calcula a altura de uma árvore binária.
- (b) Defina uma função que cria uma cópia de uma árvore.
- (c) Defina uma função que inverte uma árvore (sem criar uma nova árvore).
- (d) Considere a seguinte definição de uma função que cria uma lista ligada de inteiros a partir de uma travessia *inorder* de uma árvore binária:

```
LInt inorder (ABin a) {
    Lint r, aux;
    if (a == NULL) r = NULL;
    else { r = (LInt) malloc (sizeof (struct slist));
        r->valor = a->valor;
        r->prox = inorder (a->dir);
        aux = inorder (a->esq);
        r = concat (aux,r);
    }
    return r;
}
```

De forma a otimizar esta função vamos apresentar uma definição alternativa que não usa a função `snoc` de listas. Esta alternativa passa por usar uma função auxiliar que insere à cabeça de uma lista (inicialmente vazia) os vários elementos da árvore.

```
LInt inorder (ABin a) {
    return (inorderAcc (a, NULL));
}

LInt inorderAcc (ABin a, LInt l) {
    LInt r;
    if (a == NULL) r = l;
    else { r = (LInt) malloc (sizeof(struct slist));
          r->valor = a->valor;
          r->prox = inorderAcc (a->dir, l);
          r = inorderAcc (a->esq, r);
        }
    return r;
}
```

Apresente definições das funções `LInt preorder (ABin a)` e `LInt posorder (ABin a)` que criam listas a partir das travessias *preorder* e *posorder* de uma árvore binária sem usar a função `concat`

- (e) Apresente uma definição recursiva e outra iterativa de uma função que insere um elemento numa árvore binária de procura.
 - (f) Defina uma função não recursiva `int maior (ABin a)` que calcula o maior elemento de uma árvore binária de procura não vazia.
 - (g) Defina uma função `ABin remove (ABin a, int x)` que remove um elemento de uma árvore binária de procura.
2. Uma das aplicações mais frequentes de árvores binárias de procura é na implementação de funções finitas. Uma função finita é um conjunto de pares (*chave, informação*) em que cada chave não aparece repetida. Nos casos em que existe uma ordem sobre as chaves pode-se então usar uma árvore binária de procura (ordenada pela chave) para guardar uma destas funções finitas.

As operações que devem estar disponíveis são:

- adicionar um novo par (*chave, informação*)
- remover o par associado a uma dada *chave*
- modificar a *informação* associada a um dado par
- procura da *informação* associada a uma dada *chave*

Considere a seguinte definição para implementar funções finitas em que as chaves são *strings* e a *informação* associada a cada chave é um endereço de memória.

```
typedef struct fmap {
    char *key;
```

```

    void *info;
    struct fmap *left, *right;
} Node, *Fmap;

```

e implemente as seguintes funções:

- (a) `Fmap addPair (Fmap f, char *k, void *i)` que adiciona um novo par (k, i) à função finita f . A função deverá retornar `NULL` caso a operação não seja possível (por exemplo porque a chave k já existe em f).
 - (b) `int remove (Fmap *f, char *k, int *r)` que remove a chave k da função $*f$. A função retorna um código de erro (0 no caso de sucesso). Note que é passado como argumento à função o endereço da árvore – trata-se de uma parâmetro de entrada/saída.
 - (c) `Fmap udapte (FMap f, char *k, void *i)` que modifica a informação associada a k para i . Se a chave k ainda não existir em f , deverá ser acrescentado o par (k, i) .
 - (d) `void *lookup (Fmap f, char *k)` que calcula a informação associada a k na função f . A função deverá retornar `NULL` caso a chave k não exista.
3. Suponha que para resolver o problema descrito na secção 1 se optou por usar uma árvore binária de procura (ordenada pelo número do aluno) vez de um array.
- (a) Defina os novos tipos de dados para esta implementação.
 - (b) Apresente definições das funções `acrescentaAluno`, `procura` e `aprovados` para esta nova implementação.
Tenha o cuidado de rever os tipos destas funções nesta nova implementação.
 - (c) Defina uma função que liberte o espaço ocupado por uma destas árvores.
4. Suponha agora que se pretende implementar uma nova funcionalidade: listagem das notas finais dos alunos, por ordem crescente do seu nome. Faça as alterações necessárias para que se tenha em qualquer altura acesso ordenado (por ordem crescente do nome) à lista dos alunos.
Implemente a referida função de listagem (para o ecran).
5. Considere a seguinte definição:

```

typedef struct node {
    int t;
    int valores [N-1];
    struct node *menores [N]
    struct {
        int valor;
        struct node * menores;
    } tab [N];
} *ArvB;

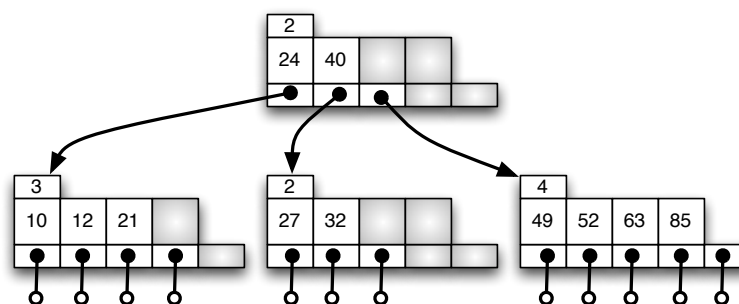
```


Esta definição pode ser usada para representar **árvores B** de ordem N .

Em cada nodo de uma destas estruturas guardam-se até $N-1$ items (neste caso, inteiros). Daí que cada nodo contenha um inteiro t que indica o número de items que estão a ser guardados nesse nodo.

Para além desse inteiro, cada nodo contém um array onde são guardados os vários items e outro onde se guardam os endereços das árvores onde se encontram os items menores ou iguais a esse item. O último elemento deste último array contém o endereço da árvore onde se encontram os items maiores do que todos os items deste nodo.

Veja-se por exemplo a seguinte árvore de altura 2 e ordem 5.



Apresente definições das seguintes funções:

- (a) `int quantos (ArvB a)` que calcula quantos elementos tem uma árvore.
- (b) `int existe (ArvB a, int x)` que testa se um dado inteiro pertence a uma árvore (retorna 0 se não existir).
- (c) `void imprime (ArvB a)` que imprime no ecran os elementos de uma destas árvores por ordem crescente.
- (d) `int maior (ArvB a)` que calcula o maior elemento de uma árvore não vazia.